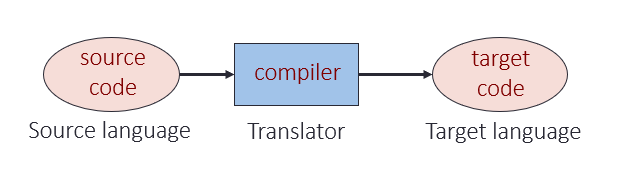
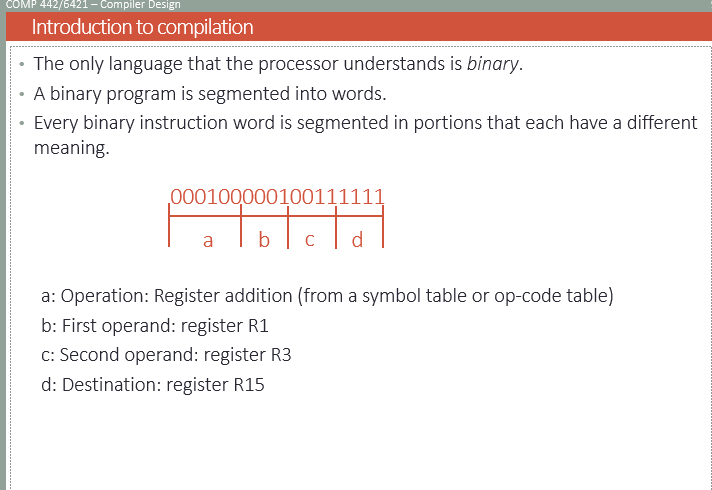
Compiler是一个translation system

他吧人类能理解的使用high level language的program转换成二进制machine language(CPU能理解)

初始source code实际上只是一串char

Compiler负责分析它的意义，并且把这个意义翻译成computer可以理解的program





处理器唯一能理解的语言是binary

Binary program被分割成不同文字

每一个被分割的binary指令文字都有自己意义

这里被分割成了abcd四个文字

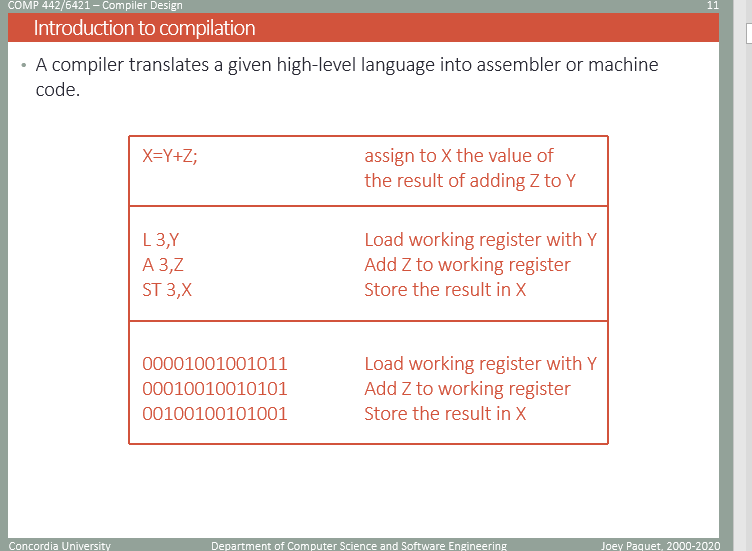
每个文字都是对内存的操作

Assembly language是higher level programming language//汇编语言

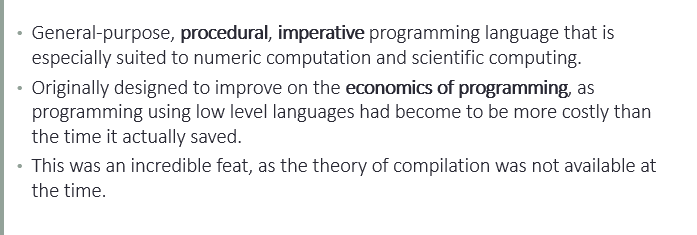


Assembly language与machine code lines一一对应且按行对应的

通过op-code table来把assembly language 转化成machine code



用户写的转成汇编语言，转成machine code



第一个compiler叫做fortran, 是一个procedural, imperative programming language用来进行数学计算或者科学计算//procedural programming是一种类别的programming，和object-oriented 这种属于同一等级

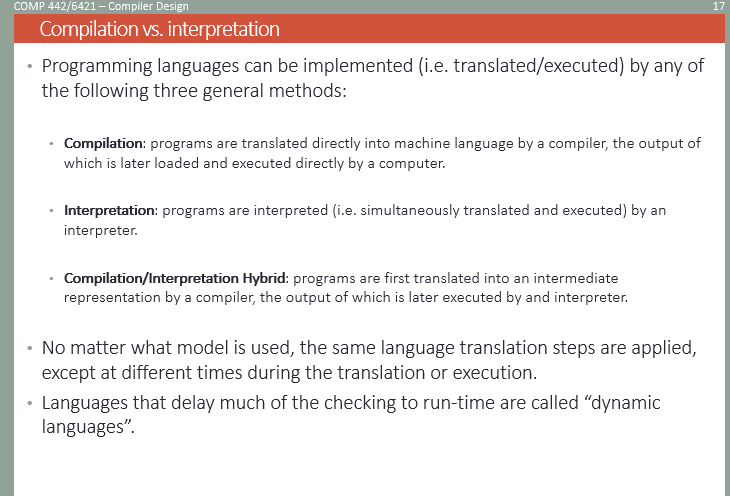
最初目的是让Programming更加具有经济效益，原来的low level 花的钱比节省的钱还多

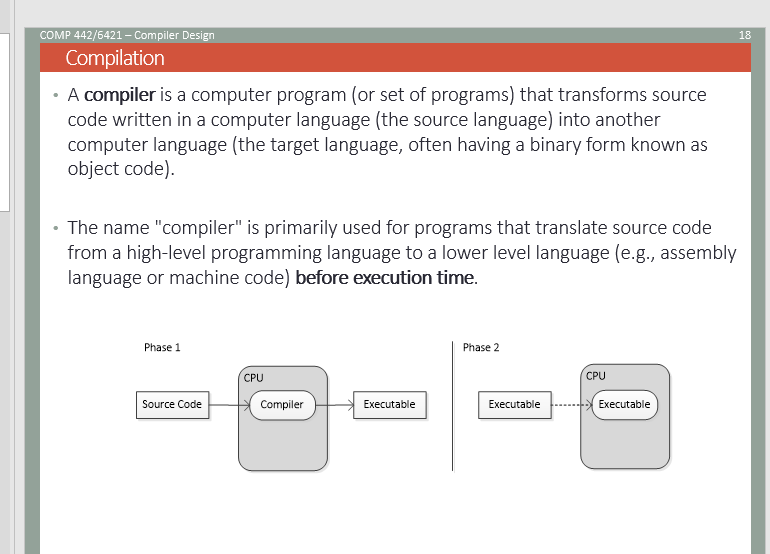
Compilation:通过compiler translate这个program，然后这个输出结果才被computer运行

Interpretation:程序被即时translate并运行通过interpreter注释器

Compilation/Interpretation hybrid混合：程序先被compiler转换成我们需要的intermediate representation，然后这个输出会被interpreter直接运行 //java就是这样。Java转换成另外一种中间语言，由java virtual machine这个interpreter运行

Dynamic language: delay对Program的分析，然后在run time的时候在check





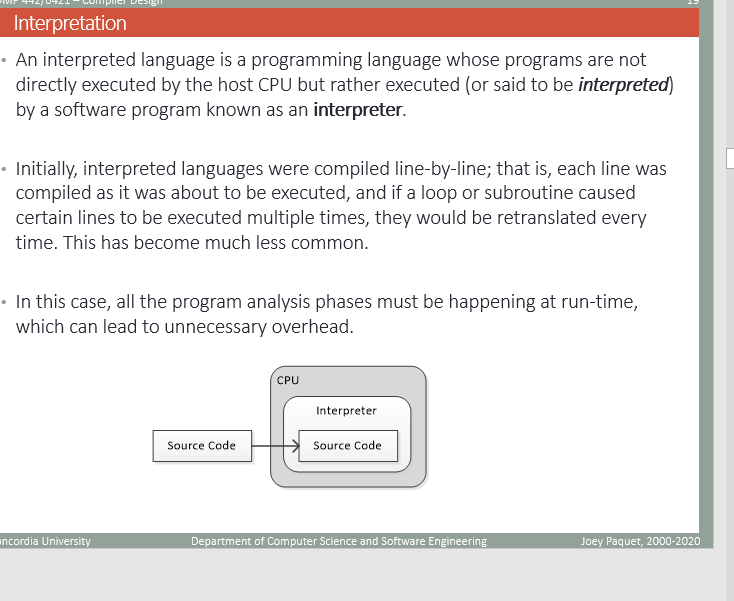
Compiler把一种computer language转换成另一种computer language，通常是source language到二进制object code

第一阶段

Source code需要通过cpu使用compiler转换成executable .exe

第二阶段

这个.exe被cpu运行



一个纯粹的interpreter：

并不是把source code直接让CPU运行，而是用CPU通过一个叫做interpreter的软件一行一行的运行source code

就直接把source code放进cpu里，一行一行的compile。这样如果有一个loop，他就会重复translate这个loop好几次，（不常见的，比较原始的interpreter）

这样就需要使用run-time的program anaylsis，增加了不必要的消耗

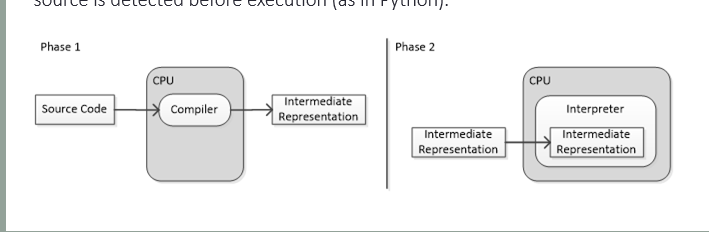
Interpreter是一个比较heavy 的program

Hybrid compilation/interpretation

在今天绝大多数interpreted language解释性语言 都使用intermediate representation中间表示来作为compilation 与interpretation的桥梁//Intermediate，调解的，中间的

在这种情况下，compiler或许会输出一段bytecode或者threaded code（反正就是一段中间代码），然后这段中间代码会被interpreter所运行。例如python,java,ruby

Intermediate representation可以被一次compile完成（java）或者每次interpreter运行的时候compile一次（ruby或perl）或每次运行前检查到源代码source code的改变后发生compile（python）.



Source code被翻译成更适合interpret的形式，通常来说是对虚拟机更容易辨别的机械语言（例如java 的bytecode）

耗时长的source code analysis部分在Interpretstion开始之前就被做掉了

有时候program error在运行前就能被检测，让execution more robust，成功率更高//robust 健康的，粗野的

在前期，language design主要取决于决定时使用compilation还是interpretation来作为cpu运行的模式

例如一些compile型语言必须明显区分data type在declare阶段，而interpreted llanguage优势在于动态，在run-time区分类别，所以这类declaration就没必要

这类语言把type binding machinism 移到run time，但这并不意外着其他程序分析阶段也必须移动到run time(复合语言)

理论上，任意语言都可以写成compile形式或者interpreted形式，所以选择哪种形式完全取决于实际经验，而不是language某些潜在的性质

有些语言既是compiler又是interpreter，例如Lisp, pascal,c,basic,phython

第二页，

Interpreting 相比compile更具有灵活性(例如不必给data type前期就指定类型)。一些特征通常也更容易安装相比于compiler：

例如：平台独立性（java的byte code）

鉴别器的重复利用

Dynamic typing以及多态

主要的interpreting的劣势在于程序运转缓慢相较于直接在cpu上转机械代码。所以我们利用一项技术”just-in time”，他吧经常运行的interppreted instruction串转化成machine code

另外一个劣势在于，由于一部分的anaylsis是在runtime完成的，所以很有可能在run time失败，所以导致program不够robust强健

Compiler’s environment

Compiler通常是linkage/translation系统的一部分

例如以c++translation过程为例